

Report No. UIUCDCS-R-87-1307

UILU-ENG 87 1780

Scheduling Periodic Jobs Using Imprecise Results

by

Jen-Yao Chung

Jane W. S. Liu

Kwei-Jay Lin

November 30, 1987

(NASA-CR-183120) SCHEDULING PERIODIC JOBS
USING IMPRECISE RESULTS (Illinois Univ.)
33 p CSCL 05A

N89-26767

Unclas
G3/81 0217512

1-8 1985

Report No. UIUCDCS-R-87-1307

Scheduling Periodic Jobs Using Imprecise Results

by

Jen-Yao Chung
Jane W. S. Liu
Kwei-Jay Lin

November 30, 1987

ACKNOWLEDGEMENT

We would like to thank Mr. Swaminathan Natarajan for his contribution in the work on the model of imprecise computations and system support for processes to return imprecise results.

This work was partially supported by the NASA Contract No. NAG 1-613 and the US Navy ONR Contract No. NVY N00014 87-K-0827.

ABSTRACT

One approach to avoid timing faults in hard, real-time systems is to make available intermediate, imprecise results produced by real-time processes. When a result of the desired quality cannot be produced in time, an imprecise result of acceptable quality produced before the deadline can be used. This paper discusses the problem of scheduling periodic jobs to meet deadlines on a system that provides the necessary programming language primitives and run-time support for processes to return imprecise results. Since the scheduler may choose to terminate a task before it is completed, causing it to produce an acceptable but imprecise result, the amount of processor time assigned to any task in a valid schedule can be less than the amount of time required to complete the task. A meaningful formulation of the scheduling problem must take into account the overall quality of the results. Depending on the different types of undesirable effects caused by errors, jobs are classified as type N or type C. For type N jobs, the effects of errors in results produced in different periods are not cumulative. A reasonable performance measure is the average error over all jobs. Three heuristic algorithms that lead to feasible schedules with small average errors are described. For type C jobs, the undesirable effects of errors produced in different periods are cumulative. Schedulability criteria of type C jobs are discussed.

I. INTRODUCTION

In a hard real-time system, a *timing fault* is said to occur when a *real-time process* delivers its result too late. A new approach, called the *imprecise computation approach*, was proposed recently as a means to avoid timing faults in hard real-time systems [1-3]. The intent of this approach is to provide timely but possibly degraded real-time services by making results that are of poorer quality available when the desired result cannot be obtained in time. Instead of providing a primary version and an alternate version of each real-time service as suggested in [4], this approach relies on making effective use of intermediate results returned by prematurely terminated real-time processes.

Specifically, in the imprecise computation approach, real-time processes are designed to have *monotone property*; a process is said to have this property if the accuracy of its intermediate result is non-decreasing as more time is spent to obtain the result. Such a process is called a *monotone process*. Practical examples of monotone processes include well-designed iterative processes and multiphase processes (e.g. see [5-10]). It is reasonable to assume that the result produced by a monotone process upon its normal termination is the desired result; this result is said to be a *precise* one. External events such as timeouts, interruptions, or crashes may cause the process to terminate prematurely. If the intermediate result produced by the process upon its premature termination is saved and made available, the application may still find the result *usable* and, hence, *acceptable*; such a result is said to be an *imprecise* one. In the imprecise computation approach, run-time system support is provided to record intermediate results produced by each real-time process at appropriate instances of the process's execution. Programming language primitives are provided to allow the programmer to specify the intermediate result variables to be recorded and the time instants to record them so that the resultant process is monotone. These primitives also allow the programmer to define a set of *error indicators*. The values of the error indicators are made available to the application process along with the intermediate results. By examining these error indicators, the application process can decide whether an intermediate result is acceptable when the desired, precise result cannot be obtained in time. A systems based on the imprecise computation approach is described in [1-2].

This paper is concerned with the problem of scheduling periodic jobs to meet deadlines on systems based on the imprecise computation approach. This problem differs from the traditional scheduling problems [11-13] for the following two reasons: (1) The scheduler may choose to terminate a task before it is completed, causing it to produce an acceptable but imprecise result. Consequently, the amounts of processor time assigned to tasks in a valid schedule can be less than the amounts of time required to complete the tasks. (2) A meaningful formulation of this problem must take into account the quality of the results. Depending on the undesirable effects caused by errors, jobs are classified as type N or type C. For type N jobs, the effects of errors in results produced in different periods are not cumulative. The average error over all jobs to be scheduled is used as the criterion of optimality. Three heuristic algorithms that lead to feasible schedules with small average errors are described, and their performance is discussed. For type C jobs, the undesirable effects of errors produced in different periods are cumulative. We are concerned with the criteria that can be used to determine whether a set of type C jobs can be scheduled to meet all deadlines.

The rest of this paper is organized as follows: Section II provides the background, definitions and assumptions that are needed in subsequent sections. Section III discusses the problem of scheduling periodic jobs to meet deadlines making use of imprecise results. The types of jobs and the appropriate performance measures are defined. Section IV describes a class of heuristic algorithms based on the rate-monotone algorithm [14] for scheduling type N jobs. Section V discusses the schedulability criteria for two special classes of type C jobs. Section VI is a summary.

II. BACKGROUND AND ASSUMPTIONS

This section defines rigorously the errors in computation results and discusses means to return imprecise results. In a system that supports imprecise computations, the intermediate results generated by each server process are recorded at appropriate instants. The exact manner in which the recordings are done is not important to us. One way is to use the process structure and the language primitives provided by Concord [1-2]. In Concord, a server type is defined for each real-time service provided by the system. Each (instance of a) server (type) is split into a

callee and a *supervisor*. A client (caller) invokes a service by sending an invocation request to the supervisor of the server providing the service. When the supervisor receives the invocation request, it initializes the callee and requests the system to schedule the callee's execution. The supervisor executes concurrently with and monitors the progress of the callee and records the intermediate results produced by the callee. The intermediate result variables to be recorded and the instants at which they are recorded are specified by the programmer using an *imprecise return* statement in the callee procedure. In particular, the supervisor records the variables specified as parameters in each imprecise return statement whenever the statement is executed. In addition to the intermediate result variables, a set of error indicators is also specified in each imprecise return statement. When the callee terminates normally, the result returned by it is passed to the client through the supervisor. If the callee terminates prematurely, the supervisor passes to the client the latest recorded values of the intermediate result variables and error indicator variables. Based on the values of error indicators the client can decide whether the intermediate result is acceptable.

In this paper, our attention is confined to the problem of scheduling periodic jobs on a tight-coupled multiprocessor system. Communication delays between clients and servers are assumed to be negligibly small. The task of assigning processors to execute invoked procedures is carried out by a scheduler. When the execution of a procedure is preempted by the scheduler to be resumed later, no result is returned to the client. A result is returned only when the callee procedure terminates either normally or involuntarily. The latter occurs when the scheduler terminates the callee procedure.

Again, we are concerned only with processes that have the *monotone property*: the precision of the result produced by the continued execution of a process with this property is monotonically non-decreasing. More specifically, we define the *correctness* C of the result R produced by a process P to be the extent to which the execution of P has progressed to produce the result. When R is not acceptable, C has the value 0; the error ϵ is 1. When R is precise, C has the value 1; ϵ is 0. Let τ be the time required for the process to terminate normally and m be the minimum time required to produce an acceptable result. When a monotone process has

executed for t units of time, the error ϵ in its result is given by

$$\epsilon(t) = \begin{cases} 1 & \text{if } t < m \\ e(t) & \text{if } m \leq t < \tau \\ 0 & \text{if } t \geq \tau \end{cases}$$

where $e(t)$ is in the range $(0, 1)$ and $e(t_1) \geq e(t_2)$ for $m \leq t_1 < t_2 \leq \tau$.

III. JOBS TYPES AND PERFORMANCE MEASURES

A workload model commonly used in studies on scheduling hard real-time jobs is the periodic-job model [14-23]. (From this point on, the terminology commonly used in studies on scheduling disciplines is used.) In this model, we are given a set $J = \{J_k\}$ of K jobs. Each job J_k is an independent unit of computation and consists of a periodic sequence of tasks $T_{k,j}$ for $j = 1, 2, 3, \dots$. The *execution time* of any task $T_{k,j}$ in J_k , denoted by τ_k , is the amount of processor time required to complete the task, i.e. for the task to execute until its normal termination. Let $\alpha_{k,j}$ be its *ready time* before which its execution cannot begin. Its *deadline* is the time instant at or prior to which it must produce an acceptable result. In particular, the deadline of the task $T_{k,j}$ is the ready time of the task $T_{k,j+1}$. Let $p_k = \alpha_{k,j+1} - \alpha_{k,j}$ be the *repetition period* of the job J_k and $r_k = 1/p_k$ be its *repetition rate*. The job J_k is, therefore, specified by the 2-tuple (p_k, τ_k) . Let p be the least common multiple of the periods p_k for $k = 1, 2, \dots, K$ and $r = 1/p$.

The problem of scheduling tasks to meet deadlines on a system that allows tasks to be terminated before they are completed, producing imprecise results, differs from the traditional scheduling problems in the following way: It is possible for the total processor time assigned to a task in a valid schedule, referred to as its *assigned time*, to be less than its execution time. Hereafter, by a (valid) *schedule* we mean one in which the assigned time of every task $T_{k,j}$ is at least equal to m_k , the minimum amount of processor time required to produce an acceptable result. We refer to m_k as the *minimal execution time* of the tasks in job J_k . A schedule in which the assigned time of every task is equal to its execution time is called a *precise schedule*. (We note that only precise schedules are valid schedules in the traditional sense.) If the assigned time,

denoted by σ , of a task is equal to its execution time τ , the error in the result produced by the task is zero. If its assigned time is less than its execution time (corresponding to involuntary termination of the task), the error $\epsilon(\sigma)$ in its result is a non-increasing function of σ . We consider here only preemptive schedules. Again, the result produced by any task is returned to the client only at the time when the task terminates, either normally when it attains τ units of processor time or involuntarily when its deadline is reached.

The term *feasible schedule* of the job set J refers to a schedule in which every task meets its deadline. In a feasible schedule, the total processor time assigned to every task prior to its deadline is equal to or larger than its minimal execution time. The problem of finding a feasible schedule of periodic jobs without taking errors in the results into consideration is a relatively easy one. — Given a job set J , we define a periodic job set $F = \{F_k\}$ in which the job F_k is specified by (p_k, m_k) . In other words, F_k has the same repetition rate as J_k but consists of tasks with execution time m_k . The tasks in F_k have the same ready times and deadlines as the corresponding tasks in J_k . The problem of finding a feasible schedule of the job set J is the same as the problem of finding a precise feasible schedule of the job set F . Known results [14–23] can be applied directly here.

A more meaningful formulation of the scheduling problem on a system that supports imprecise computations must take into consideration the quality of the results produced by the tasks. For this reason, we want to find algorithms leading to feasible schedules that keep some cost functions of errors small.

To motivate our choices of performance measures, we note that for different types of jobs, errors in the results cause different undesirable effects. As an example, consider a job consisting of tasks that periodically receive, enhance, and transmit frames of video images. The effect of the error in the imprecise result generated in a period is a reduction in the quality of the transmitted image in that period. This undesirable effect is tolerable for many applications. Hence, no timing fault occurs as long as sufficient processor time is assigned to every task before its deadline so that a frame of acceptable quality is transmitted on time. For applications such as this one, errors in the results produced in different periods do not have cumulative effect. We

refer to jobs of this type as *type N(on-cumulative) jobs*.

A reasonable performance measure for type N jobs is the average error of all results. Given a feasible schedule of the job set J , the average error is defined as follows: Let the error in the result produced by the task $T_{k,j}$ be denoted by $\epsilon_k(\sigma_{k,j})$ when its assigned time is $\sigma_{k,j}$. Let Q_k be the number of consecutive periods over which the average error of job J_k is computed. At any time s beyond the deadline of the $(i + Q_k - 2)$ th task but before the deadline of the $(i + Q_k - 1)$ th task in J_k for some i , the average error of J_k is

$$E_k = [\sum_{j=i}^{i+Q_k-2} \epsilon_k(\sigma_{k,j}) + \epsilon_k(\sigma_{k,c})] / Q_k$$

where $\sigma_{k,c}$ is the processor time attained by the current task $T_{k,i+Q_k-1}$ at time s . Let $E_k(s-)$ be the contribution to the average error of J_k by errors in the results produced during the $Q_k - 1$ periods prior to s . This equation can be rewritten as

$$E_k = E_k(s-) + \epsilon_k(\sigma_{k,c}) / Q_k \quad (1)$$

We may choose to compute the average error over a time interval of duration qp , an integer q times the least common multiple p of all repetition periods. In this case, the average error of J_k is approximately equal to

$$E_k = \frac{p_k}{qp} \sum_{j=l}^{l+qp r_k - 1} \epsilon_k(\sigma_{k,j}) \quad (2)$$

The average error over all jobs in J is

$$E = \sum_{k=1}^K w_k E_k \quad (3)$$

where w_k are non-negative constant weights and $\sum_{k=1}^K w_k = 1$. (These weights reflect the relative importance of different jobs.) Given a set of type N jobs, we want to find feasible schedules that have the minimum average error E among all feasible schedules of J .

As an example of a different type of applications, suppose that we have a periodic job in which each task processes the radar signal returned from a tracked target and generates the

coordinates and the velocity of the target for display purposes. When a task terminates prematurely, it produces coarse estimates of the target position and velocity in that period. Typically, it is essential that a precise result be obtained every now and then. For example, the position of the target must be accurately displayed every 30 seconds. Hence, if the results produced by the position computation tasks in several consecutive periods are imprecise, the task in the next period must complete normally and produce a precise result. Otherwise, a timing fault occurs. We refer to jobs of this type as *type C (umulative) jobs*. A reasonable formulation of the scheduling problem for type C jobs is as follows: Let the cost $\beta(\epsilon_k(\sigma_{k,j}))$ be a monotone non-decreasing function of the error $\epsilon_k(\sigma_{k,j})$. At any time s between the deadlines of $T_{k,n}$ and $T_{k,n+1}$, let the cumulative cost due to errors of J_k be

$$\gamma_k = \sum_{j=l+1}^n \beta(\epsilon_k(\sigma_{k,j})) + \beta(\epsilon_k(\sigma_{k,c})) \quad (4)$$

where $T_{k,l}$ ($l < n$) was the last task in J_k that terminated normally and produced zero error, and $\sigma_{k,c}$ is the processor time attained by the current task $T_{k,n+1}$ at time s . Clearly, a timing fault occurs whenever the assigned time of any task is less than m_k . A timing fault occurs also when the cumulative cost γ_k exceeds an upper limit and the deadline of current task $T_{k,n+1}$ is reached before it attains τ_k units of processor time. The scheduling problem we want to consider is: given a set of jobs, find a schedule in which there is no timing fault.

IV. HEURISTIC ALGORITHMS FOR TYPE N JOBS

This section describes three heuristic algorithms for type N jobs and discusses their performance. These algorithms are based on the rate-monotone algorithms [14,15]. Specifically, we first use either the rate-monotonic next-fit algorithm or the rate-monotonic first-fit algorithm [15] to assign individual jobs in the given set J to the processors in a multiprocessor system; jobs in J are sorted in the order of increasing period lengths and are assigned to processors on next-fit or first-fit basis. In deciding whether a job can be assigned to (i.e. can fit on) a processor, we use the minimal execution time of its tasks instead of the execution time to compute the utilization factors as follows: Let the *minimal utilization factor* of the job J_k be $u_k = r_k m_k$. Suppose that n jobs with a total minimal utilization factor u are already assigned to

a processor. If an addition job with repetition rate r , execution time τ , and minimal execution time m is assigned to this processor, the total minimal utilization factor of the $n + 1$ jobs is $u + r m$. This job is assigned to the processor only if

$$u + r m \leq (n + 1) (2^{1/(n+1)} - 1)$$

We note that these $n + 1$ jobs are not precisely schedulable on the processor in general. However, they are schedulable if the assigned time of every task is equal to its minimal execution time [14].

We now assume that jobs are assigned to processors as described above and confine our attention to the problem of scheduling periodic jobs on a single processor to meet all deadlines while keeping the average error of all jobs on the processor small. The mixed scheduling algorithms described below use the rate-monotone algorithm to ensure that all deadlines are met but use different strategies to improve result quality. They work as follows: Given a set J of K jobs with a total minimal utilization factor equal to or less than $K(2^{1/K} - 1)$ to be scheduled on a processor, we define two job sets, the *first set* F and the *last set* L . For each job $J_k = (p_k, \tau_k)$ in J , there is a job $F_k = (p_k, m_k)$ in the first set F . In particular, the task $T_{k,j}(F)$ in the job F_k is the first portion of the task $T_{k,j}$ in J_k , and the execution time of $T_{k,j}(F)$ is m_k for all j . Similarly, for each job J_k in J , there is a job $L_k = (p_k, \tau_k - m_k)$ in the last set L . The task $T_{k,j}(L)$ in the job L_k is the last portion of the task $T_{k,j}$ in J_k , and the execution time of $T_{k,j}(L)$ is $\tau_k - m_k$. Furthermore, the ready times and deadlines of the tasks $T_{k,j}$, $T_{k,j}(F)$, and $T_{k,j}(L)$ are the same. Instead of the job set J , we schedule the sets F and L using the following preemptive, priority-scheduling strategy.

Theorem IV-1: All jobs in F are assigned higher priorities than jobs in L and are precisely scheduled according to the rate-monotone algorithm. This strategy of scheduling jobs in J guarantees that all deadlines are met regardless how tasks in L are scheduled.

Proof: Because

$$\sum_{k=1}^K r_k m_k \leq K(2^{1/K} - 1)$$

this strategy ensures that all tasks in F complete before their deadlines independent of how tasks

in L are scheduled [14]. For every schedule of F and L obtained in this manner, there is an equivalent schedule of jobs in J in which the assigned time of every task is at least equal to its minimal execution time. Therefore, the theorem follows. ■

Figure 1 shows an example in which the job set J consists of 4 jobs. They are $(2, 1)$, $(4, 0.5)$, $(5, 0.5)$ and $(6, 1.5)$ and have minimal execution times 0.5, 0.2, 0.1 and 1.0, respectively. The total utilization factor of the job set J is equal to 0.975. J is not precisely schedulable

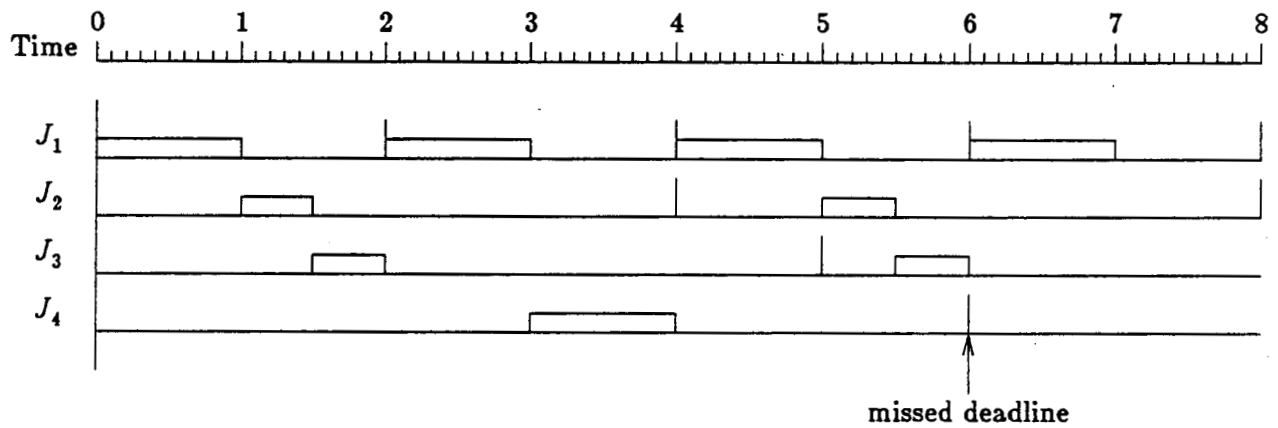


Figure 1a. A precise rate-monotone schedule

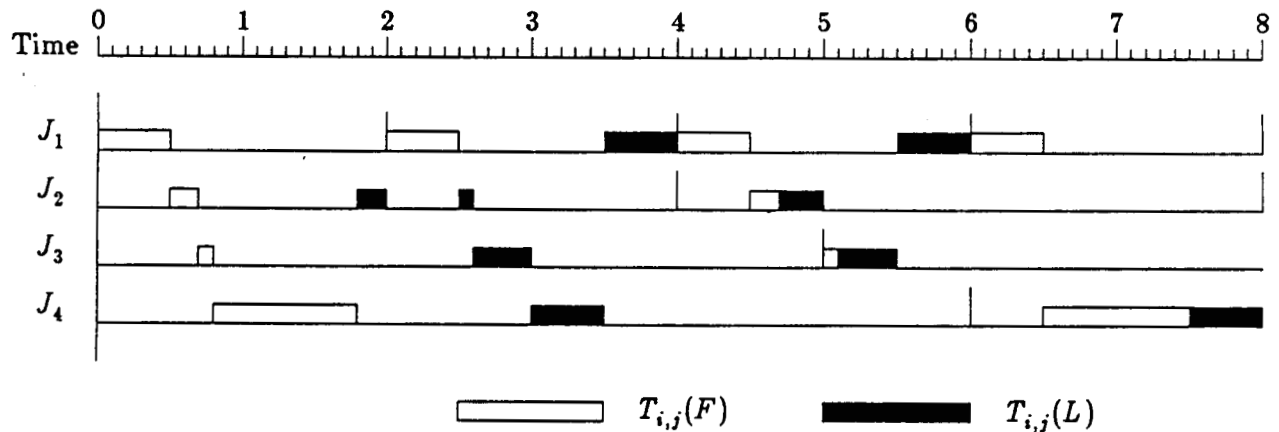


Figure 1b. A feasible schedule — an example on scheduling of type N jobs.

according to the rate-monotone algorithm as shown in Figure 1a. (The deadline of $T_{4,1}$ is missed at time 6.) However, the first set F consists of (2, 0.5), (4, 0.2), (5, 0.1) and (6, 1.0) with a total utilization factor 0.4867; it is guaranteed to be precisely schedulable according to the rate-monotone algorithm. Figure 1b shows a rate-monotone schedule of F . Black bars in Figure 1b indicate the time intervals during which the spared processor time is assigned to jobs in the last set L , consisting of (2, 0.5), (4, 0.3), (5, 0.4) and (6, 0.5).

In general, the fraction of processor time used to execute tasks in F is at most equal to $K(2^{1/K} - 1)$ (e.g, 0.82 for $K=1$ and $\ln 2$ for large K). Whenever there is no task in F to be executed, the spared processor time can be used to execute tasks in L to improve the quality of the results. The algorithms described below differ in how priorities are assigned to jobs in L .

IV.1. The Least Utilization Algorithm

To see how priorities should be assigned to jobs in L so that the average error is kept small, let us first consider the special case of linear error functions given by

$$\epsilon_k(\sigma_{k,j}) = \begin{cases} 0 & \text{if } \tau_k < \sigma_{k,j} \\ 1 - (\sigma_{k,j} - m_k) / (\tau_k - m_k) & \text{if } m_k \leq \sigma_{k,j} \leq \tau_k \\ 1 & \text{if } 0 \leq \sigma_{k,j} < m_k \end{cases} \quad (5)$$

From Eqs. (2) and (3), the average error of the job set J over a time interval of length $q p$ is given by

$$E = 1 - \frac{1}{q p} \sum_{k=1}^K \frac{w_k}{V_k} \sigma_k(L) \quad (6)$$

where

$$\sigma_k(L) = \sum_{j=l}^{l+qpr_k-1} \sigma_{k,j} - q p r_k m_k \quad (7a)$$

is the total processor time assigned to all tasks in L_k over qpr_k periods, and

$$V_k = (\tau_k - m_k) / p_k \quad (7b)$$

Without loss of generality, let the jobs be indexed so that

$$w_1 / V_1 > w_2 / V_2 > \cdots > w_K / V_K \quad (8)$$

Since $0 \leq \sigma_k(L)$ for $k = 1, 2, \dots, K$ and $\sum_{k=1}^K \sigma_k(L) \leq q p (1 - \sum_{k=1}^K r_k m_k)$, the sum in the right hand side of Eq. (6) is maximized (or E is minimized) when $\sigma_1(L)$ is made as large as possible. For a given value of $\sigma_1(L)$, E is minimized by making $\sigma_2(L)$ as large as possible and so on. This suggests the following algorithm, referred to as the *least utilization algorithm*. It is preemptive and priority-driven. Given a job set J , we divide it into the first set F and the last set L as defined above. All jobs in F are assigned higher priorities than the jobs in L . Priorities of the individual jobs in F are assigned statically on rate-monotone basis, i.e., jobs with higher repetition rates are assigned higher priorities. Priorities of jobs in L are assigned also statically but according to their values of the ratio w_k / V_k ; the larger this ratio is, the higher the priority. (When all jobs in J have the same constant w_k and minimal execution time m_k , jobs with smaller utilization factors are assigned higher priorities and, hence, the name of this algorithm.) At any time when there is no task in the first set F to be executed, the processor is assigned to jobs in the last set L on preemptive, priority-driven basis with L_1 having higher priority than L_2 , L_2 having higher priority than L_3 and so on.

For illustrative purpose, we consider the example shown in Figure 1. Suppose that $w_k = 1/4$ for $k = 1, 2, 3$ and 4 and $q = 1$. Since $V_1 = 0.25$, $V_2 = 0.075$, $V_3 = 0.08$ and $V_4 = 0.083$, the priorities assigned to L_1 , L_2 , L_3 and L_4 according to the least utilization algorithm are 4, 1, 2 and 3, respectively, with a smaller number denoting a higher priority. The total assigned time to jobs L_1 , L_2 , L_3 and L_4 in $p = 60$ units of time are $\sigma_1(L) = 10.0$, $\sigma_2(L) = 4.5$, $\sigma_3(L) = 4.8$ and $\sigma_4(L) = 5.0$. The average error over 60 units of time is 0.0833.

Let

$$U = \sum_{k=1}^K r_k r_k \quad (9a)$$

$$u = \sum_{k=1}^K m_k r_k \quad (9b)$$

be the total utilization factors of the jobs in J and F , respectively. Let the jobs be indexed so that (8) is true. When $U > 1$, there exists an integer i such that $1 \leq i < K$ and

$$\sum_{k=1}^i r_k r_k \leq 1 < \sum_{k=1}^{i+1} r_k r_k \quad (10)$$

The average error has the following lower bound when the error functions are linear as given by Eq. (5) and the least utilization algorithm is used.

Theorem IV-2:

$$E \geq \begin{cases} 1 - \sum_{k=1}^i w_k + \frac{w_{i+1}}{V_{i+1}} \left(\sum_{k=1}^i V_k - 1 + u \right) & \text{if } U > 1 \\ 0 & \text{if } U \leq 1 \end{cases} \quad (11)$$

where the index i is defined by (10). Furthermore, this bound is tight.

Proof: In a time interval of length $q p$ units, the processor is assigned to jobs in the first set F for $q p \sum_{k=1}^K m_k r_k = q p u$ units. $q p (1 - u)$ units of processor time are available to be assigned to jobs in the last set L . We consider the following two cases:

(i) $U > 1$. In this case, an optimal assignment is one in which the assigned time of every task in L_k is $r_k - m_k$ for $k = 1, 2, \dots, i$ and the remaining processor time is assigned to tasks in the job L_{i+1} . For this assignment,

$$\sigma_k(L) = q p r_k (\tau_k - m_k) \quad \text{for } k = 1, 2, \dots, i$$

$$\sigma_{i+1}(L) = q p \left[1 - u - \sum_{k=1}^i r_k (\tau_k - m_k) \right]$$

From Eqs. (6) and (7b) we have

$$E \geq 1 - \frac{1}{q p} \left\{ \sum_{k=1}^i \frac{w_k}{V_k} q p r_k (\tau_k - m_k) + \frac{w_{i+1}}{V_{i+1}} q p \left[1 - u - \sum_{k=1}^i r_k (\tau_k - m_k) \right] \right\}$$

$$= 1 - \sum_{k=1}^i w_k + \frac{w_{i+1}}{V_{i+1}} \sum_{k=1}^i V_k - \frac{w_{i+1}}{V_{i+1}} (1-u)$$

(ii) $U \leq 1$. An optimal assignment is one in which the assigned time of all tasks in L_k is $\tau_k - m_k$ for all $k = 1, 2, \dots, K$. In other words,

$$\sigma_k(L) = q p r_k (\tau_k - m_k)$$

for all k . From Eqs. (6) and (7b)

$$E \geq 1 - \frac{1}{q p} \sum_{k=1}^K \frac{w_k q p}{V_k} r_k (\tau_k - m_k) = 0$$

In both cases, the equality holds for any set of jobs in which all jobs have the same repetition period p . The case of $U > 1$ is illustrated by Figure 2. ■

In the special case of K identical jobs with $w_k = 1/K$ and $V_k = r_k (\tau_k - m_k) = (U - u)/K$, we have

Corollary IV-1:

$$E = \begin{cases} (U-1)/(U-u) & \text{if } U > 1 \\ 0 & \text{if } U \leq 1 \end{cases}$$

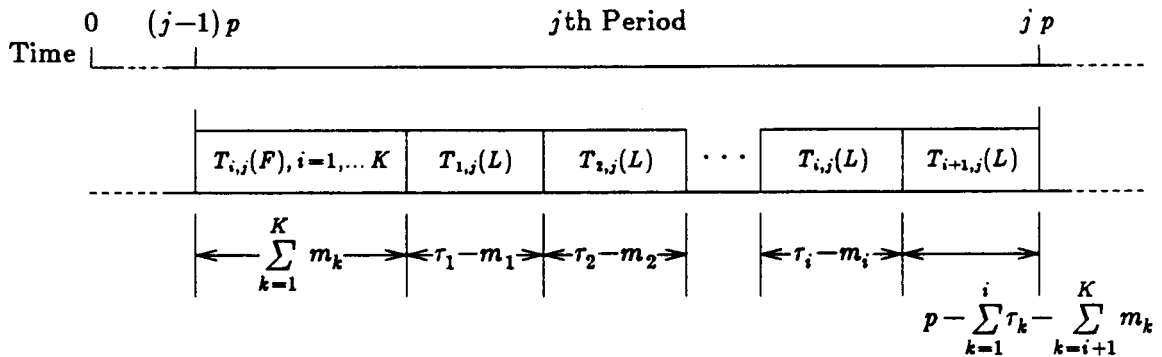


Figure 2. An example on a set of jobs for which bound (11) is tight.

Proof: We consider the case when $U > 1$ in the following; for $U \leq 1$ it is obviously that $E = 0$. From Eq. (6) and Theorem IV-2, we have

$$E = 1 - \frac{1}{q p} \frac{1/K}{(U-u)/K} q (p - \sum_{k=1}^K m_k) = 1 - \frac{1-u}{U-u} = \frac{U-1}{U-u}$$

This average error is achieved by assigning all tasks in L equal amount of processor time. ■

Corollary IV-1 states that as long as $U \leq 1$, the least utilization algorithm can schedule all identical jobs to meet deadlines while keeping the average error zero. In other words, the least utilization algorithm is optimal, like the earliest deadline first algorithm [14], for identical jobs. However, the earliest deadline first algorithm cannot respond dynamically to change in load conditions; sudden increases in the execution time of any job and additions of new jobs tend to cause timing faults when U is close to 1. The least utilization algorithm can respond to these changes as long as the first set remains to be schedulable according to the rate-monotone algorithm.

Unfortunately, the spared processor time that is available after all ready tasks in F are assigned is not always be used in an optimal manner by the least utilization algorithm to reduce average error. An example illustrating this situation is shown in Figure 3. In this example, there are three jobs, $J_1 = (3, 0.9)$, $J_2 = (2, 0.8)$ and $J_3 = (1, 0.6)$. The minimal execution time of all

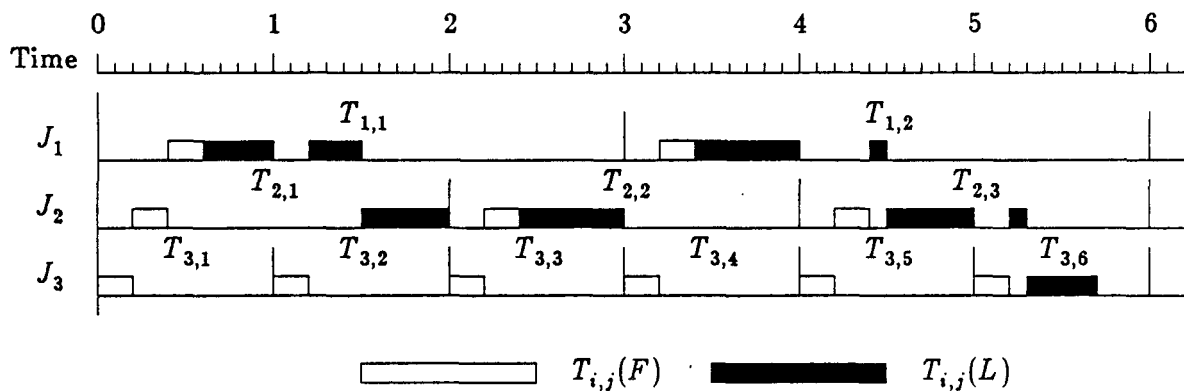


Figure 3. An example on suboptimal use of processor time to reduce average error.

jobs is 0.2 and $V_1 = 0.233$, $V_2 = 0.3$ and $V_3 = 0.4$. The last set L consists of $(3, 0.7)$, $(2, 0.6)$ and $(1, 0.4)$, and the priorities are 1, 2 and 3, respectively. Figure 3 shows a schedule obtained using the least utilization algorithm. Note that $T_{2,1}(L)$ is not completed and only $T_{3,6}(L)$ among all $T_{3,j}(L)$'s is assigned any processor time. Yet the processor is left idle during the last 0.3 units of time before the deadline of $T_{3,6}$.

In general, the worst case average error depends on the parameters of the jobs. To illustrate how the worst case average error depends on these parameters without getting hopelessly lost in complicated notations, we consider here the *simply periodic* case [24]. A set of jobs J is said to be simply periodic if every repetition rate is an integral multiple of the next smaller rate (and thus of all smaller rates).

Theorem IV-3: Consider a set J of simply periodic jobs with $U > 1$. Let the jobs be indexed so that (8) is satisfied. When $p = p_1 > p_2 > \dots > p_K$,

$$s_i = p_i - \sum_{k=i}^K \frac{p_i}{p_k} m_k \geq r_i - m_i \quad i = 1, 2, \dots, K \quad (12)$$

the error functions are linear as given by Eq. (5), and the least utilization algorithm is used,

$$E \leq 1 - \sum_{k=1}^i w_k + \frac{w_i p_i}{p_{i-1}} \left\lceil \frac{1}{s_i} \sum_{k=1}^i r_k \right\rceil$$

where $\lceil x \rceil$ denotes the smallest integer that is equal to or larger than x , and i is the smallest index satisfying inequality

$$s_i - \sum_{k=1}^{i-1} r_k \leq r_i - m_i$$

Proof: Let $\underline{\sigma}_k(L)$ denote the worst case total assigned time of the tasks $T_{k,j}(L)$ in L_k during the time interval of length $q p$ over which the average error is computed. To find $\underline{\sigma}_k(L)$, we examine the first p_1/p_k periods of J_k for $k = 1, 2, \dots, K$.

During the first period (and subsequent periods) of J_1 of length p_1 , the amount of spared processor time S_1 after completing all tasks in the first set is s_1 as defined in (12) since there are

p_1/p_k ready tasks in F_k in this period. Because L_1 has the highest priority among all L_k 's, we have $\underline{\sigma}_1(L) = q(\tau_1 - m_1) = q p V_1$.

Because of the way in which priorities are assigned, at least p_2/p_k tasks in F_k , one task in F_1 and one task in L_1 are completed before any processor time is assigned to the task $T_{2,1}(L)$ in L_2 during the first period of J_2 . The spared processor time during this period that can be assigned to $T_{2,1}(L)$ is $S_2 = s_2 - \tau_1$. There are the following three cases: (i) If $S_2 \geq \tau_2 - m_2$, we have $\underline{\sigma}_2(L) = q p V_2$. (ii) If $0 < S_2 < \tau_2 - m_2$, all S_2 units of processor time in the first period of J_2 is assigned to $T_{2,1}(L)$. The assigned time of the tasks $T_{2,j}(L)$ is $\tau_2 - m_2$ during the subsequent $(p_1/p_2 - 1)$ periods. Hence

$$\sigma_2(L) = q \left[s_2 - \tau_1 + \left(\frac{p_1}{p_2} - 1 \right) (\tau_2 - m_2) \right] \quad (13)$$

(iii) If $S_2 < 0$, the execution time of $T_{1,1}(L)$ is larger than $s_2 - m_1$. There is no spared processor time available to be assigned to the tasks in L_2 during its first $\lceil \tau_1/s_2 \rceil - 1$ periods.

$$\sigma_2(L) = q \left[\lceil \tau_1/s_2 \rceil s_2 - \tau_1 + \left(\frac{p_1}{p_2} - \lceil \tau_1/s_2 \rceil \right) (\tau_2 - m_2) \right]$$

(Note that Eq. (13) is a special case of this expression when $\lceil \tau_1/s_2 \rceil = 1$.) Moreover,

$$\sigma_2(L) > \underline{\sigma}_2(L) = q \left(\frac{p_1}{p_2} - \lceil \tau_1/s_2 \rceil \right) (\tau_2 - m_2)$$

and $\underline{\sigma}_k(L) = 0$ for $k = 3, 4, \dots, K$.

Suppose that the spared processor time during every period of L_{i-1} after completing all ready tasks in F and L_1, L_2, \dots, L_{i-2} is larger than $\tau_{i-1} - m_{i-1}$, i.e.,

$$S_{i-1} = s_{i-1} - \tau_1 - \tau_2 - \dots - \tau_{i-2} > \tau_{i-1} - m_{i-1}$$

The assigned time of every task $T_{k,j}(L)$ is $\tau_k - m_k$ for $k = 1, 2, \dots, i-1$. However

$$S_i = s_i - \tau_1 - \tau_2 - \dots - \tau_{i-1} < \tau_i - m_i$$

We have

$$\underline{\sigma}_k(L) = \begin{cases} q p V_k & k=1, 2, \dots, i-1 \\ \frac{q p}{p_{i-1}} \left(\frac{p_{i-1}}{p_i} - \left\lfloor \frac{1}{s_i} \sum_{j=1}^{i-1} \tau_j \right\rfloor \right) (\tau_i - m_i) & k=i \\ 0 & k=i+1, \dots, K \end{cases}$$

Hence

$$\begin{aligned} E &\leq 1 - \frac{1}{q p} \sum_{k=1}^{i-1} \frac{w_k}{V_k} q p V_k - \frac{1}{q p} \frac{w_i}{V_i} \frac{q p}{p_{i-1}} \left(\frac{p_{i-1}}{p_i} - \left\lfloor \frac{1}{s_i} \sum_{j=1}^i \tau_j \right\rfloor \right) (\tau_i - m_i) \\ &= 1 - \sum_{k=1}^i w_k - \frac{w_i}{V_i} \frac{\tau_i - m_i}{p_{i-1}} \left\lfloor \frac{1}{s_i} \sum_{k=1}^i \tau_k \right\rfloor \end{aligned}$$

■

We note that the inequalities in (12) implies that every task of L_i can be precisely scheduled if L_i is the only job in the last set L . This restriction places a reasonable upper limit on the total minimum utilization factor and the utilization factors of individual jobs. The bound in Theorem IV-3 can be applied to the more general case when periods of jobs in J do not have distinct values if jobs with the same repetition period have the same weighted utilization factor w_k / V_k . Specifically, suppose that jobs $J_{i_1}, J_{i_2}, \dots, J_{i_j}$ have the same period p_i and weighted utilization factor w_i / V_i . These jobs can be clustered together as one job with minimal execution time $m_i = m_{i_1} + m_{i_2} + \dots + m_{i_j}$ and execution time $\tau_i = \tau_{i_1} + \tau_{i_2} + \dots + \tau_{i_j}$.

IV.2. Other Heuristic Algorithms

In general, the underlying procedure may not converge linearly, and the error in the result produced by a task does not decrease linearly as its execution continues. We consider two additional algorithms that keep the average error small for general error functions. Again, they differ from the least utilization algorithm only in how priorities are assigned to jobs in the last set L .

According to the *least attained time algorithm*, priorities are assigned to jobs in L dynamically in the following manner: At any time when the processor is free from executing tasks in F , the highest priority is assigned to the job whose current task has the least attained processor time. In other words, each ready task in the last set L is scheduled together with other ready tasks in L on the shortest-elapse-time-first basis until it is complete or its deadline is reached whichever is sooner. Many iterative procedures converge faster during earlier iterations than later iterations; the error decreases faster during earlier iterations and slower during later iterations. For this type of error functions, the least attained time algorithm should yield smaller average error than the least utilization algorithm.

To see how the performance of the least attained time algorithm depends on the behavior of the error function ϵ_k , we examine the average error of J_k given by Eq. (1) at time s when the attained processor time of the current task of J_k is less than its execution time, i.e. $\sigma_{k,c} < \tau_k$. If a small increment of processor time of duration δ is assigned to this task, the average error of J_k is approximately given by

$$E_k = E_k(s-) + [\epsilon_k(\sigma_{k,c}) + \epsilon_k'(\sigma_{k,c})\delta + \frac{1}{2}\epsilon_k''(\sigma_{k,c})\delta^2] / Q_k$$

where $\epsilon_k'(x)$ and $\epsilon_k''(x)$ are the first and second derivatives of the function $\epsilon_k(x)$, respectively. Due to monotone property, the former is always negative. However, for a large class of error functions of practical interest, the second derivative of $\epsilon_k(\sigma_{k,c})$ is positive, i.e., the error functions are convex. In this case, the performance of the least attained time algorithm depends on the relative values of these derivatives and the ratio w_k / Q_k . An algorithm that assigns higher priorities to jobs with larger values of

$$\frac{w_k}{Q_k} \left[\epsilon_k'(\sigma_{k,c})\delta + \frac{1}{2}\epsilon_k''(\sigma_{k,c})\delta^2 \right]$$

is likely to yield the smallest average error. However, this algorithm is not a practical one. The behavior of the error functions is typically not known, making it impossible to compute their derivatives statically. Computing the derivatives on dynamically basis would introduce unacceptably high overhead.

When the behavior of the error function is not known, we may want to use an algorithm that attempts to keep the average error small by making good use of slack time between the time instant at which the task in the current period first attains m_k units of processor time to the beginning of the next period, the ready time of the next task. The lower bound on slack times derived in [19] provides the basis of the *best slack time algorithm*. It has been shown that after the completion of the task $T_{k,j}(F)$, the slack time to the ready time of the task $T_{k,j+1}(F)$ is at least equal to the $0.207 q_{k,j}$ where $q_{k,j}$ is the length of the last quantum of processor time assigned to the task $T_{k,j}(F)$. This lower bound on slack times of individual jobs allows us to bound the length of time between the instant when the processor becomes free from executing tasks in F and the ready time of the next task in F . When the processor is free to execute tasks in L , the best slack time algorithm assigns the highest priority to the job with the largest slack time.

V. SCHEDULING TYPE C JOBS

We consider here a case of practical interest where the cost function $\beta(x)$ in Eq. (4) is equal to 1 for $x > 0$ and is equal to zero for $x = 0$. For this cost function, requiring that the cumulative cost remains under an acceptable upper limit is the same as requiring that at least one task among the tasks $T_{k,j}$ in several consecutive periods of each job J_k be completed normally and produces a zero error. Specifically, a schedule for such a set of type C jobs is said to be a *feasible schedule* if (i) the assigned time of every task in J_k is at least equal to m_k and (ii) the assigned time of at least one task among every Q_k consecutive tasks in J_k is equal to τ_k . We refer to Q_k as the *cumulation rate* of J_k . A set J of (type C) jobs is said to be schedulable if there exists a feasible schedule of J .

V.1. Scheduling Jobs with the Same Repetition Period and Cumulation Rate

We consider now the special case when all jobs in the given set J have the same repetition period p and the same cumulation rate $2Q - 1$. Given such a set of K jobs, we define a first set F as in Section III; for each job $J_k = (p, \tau_k)$ in J , there is a job $F_k = (p, m_k)$ in F . Figure 4 illustrates the problem of scheduling the jobs in J . During each period, $\sum_{k=1}^K m_k$ units of processor

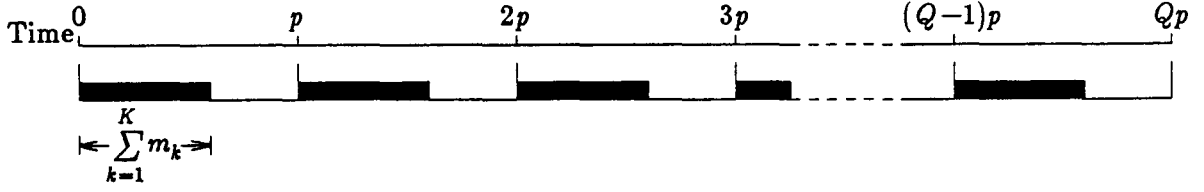


Figure 4. Scheduling of type C jobs with the same repetition period.

time are assigned to the current tasks of F_k for all k . $p - \sum_{k=1}^K m_k$ units of processor time are available each period and can be used to complete some tasks in J . Let $R = \{R_k\}$ denote a set of K independent tasks defined as follows and is referred to as the *remaining set*: for each job in J , there is a task with execution time $(\tau_k - m_k)$ in R . We note the following.

Theorem V-1: The problem of finding a feasible schedule of the job set J with repetition period p and cumulation rate $2Q - 1$ is the same as finding a non-preemptive, precise schedule of the remaining set R on Q processors with completion time equal to or less than $p - \sum_{k=1}^K m_k$.

Proof: We segment the time into intervals of Q periods each. For every job J_k , if one task is completed normally in each interval, i.e. has assigned time τ_k , then at least one task in each consecutive $2Q - 1$ periods is completed.

To find a feasible schedule, we assign the first $\sum_{k=1}^K m_k$ units of processor time at the beginning of each period to the current tasks in the first set. To complete one task in every job J_k every Q periods, the remaining portion of the task must be scheduled precisely in one of the Q periods. The execution time of this portion is $\tau_k - m_k$. The statement of the theorem follows. ■

Since the problem of finding optimal, non-preemptive, precise schedules of independent tasks to meet uniform deadline on Q processors is NP-complete [25], it follows from Theorem V-1 that the problem of finding optimal feasible schedules (e.g. with the smallest possible cumulation rate) of type C jobs with identical repetition period and cumulation rate is also NP-

complete. We consider now the following heuristic algorithm, called the *length monotone algorithm*: In the beginning of every period, the current task in F_k is assigned m_k units of processor time for all $k = 1, 2, \dots, K$. The remaining set R is then scheduled precisely in the Q intervals, each of length $p - \sum_{k=1}^K m_k$, using the *first-fit-decreasing (FFD)* algorithm [26]. In other words, the tasks in R are sorted so that their execution times $\tau_k - m_k$ are in decreasing order; these tasks are assigned to the Q intervals on first-fit-decreasing basis.

Figure 5 shows an example in which the set J consists of 5 jobs. They are $(1, 0.4)$, $(1, 0.5)$, $(1, 0.3)$, $(1, 0.4)$ and $(1, 0.2)$, and all have minimal execution times 0.1 and the cumulation rate 5 (i.e., $Q = 3$). During each period 0.5 unit of processor time is assigned to the current tasks of F and 0.5 unit of processor time is available for the remaining set R . The execution times of the tasks in R are 0.3, 0.4, 0.2, 0.3 and 0.1, respectively. According the length monotone algorithm the tasks R_k are scheduled at time 1.5, 0.5, 1.8, 2.5 and 0.9, respectively.

Lemma V-1: The tasks in the remaining set R can be scheduled precisely in Q intervals of length equal to or less than

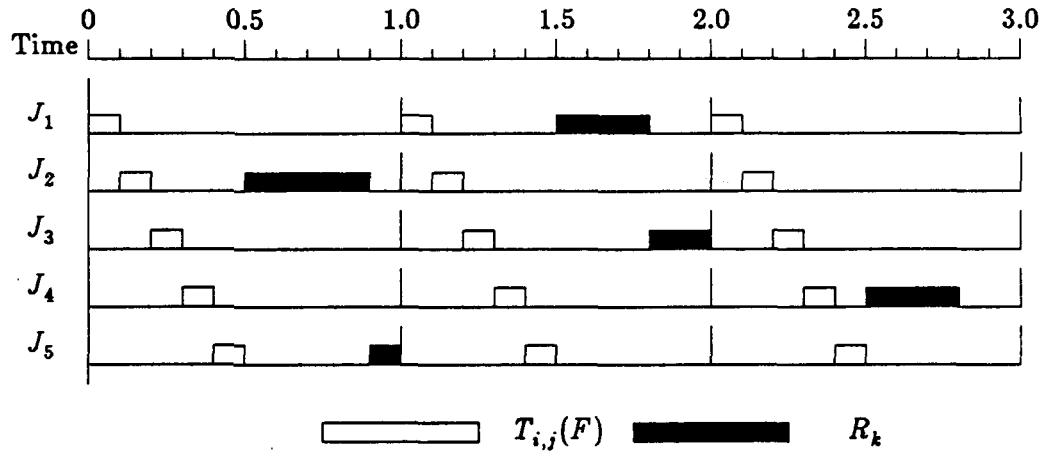


Figure 5. An example on scheduling of type C jobs using length monotone algorithm.

$$C_U = \max \left\{ \frac{2}{Q+1} \sum_{k=1}^K (\tau_k - m_k), \max_{1 \leq k \leq K} (\{\tau_k - m_k\}) \right\}$$

using the FFD algorithm. Moreover, this bound is tight.

Proof: Suppose that the remaining set cannot be scheduled precisely in Q intervals of length equal to C_U using the FFD algorithm. Without loss of generality, let

$$\tau_1 - m_1 \geq \tau_2 - m_2 \geq \dots \geq \tau_K - m_K$$

Let R_j be the first task in \mathbf{R} that cannot be fit in the Q intervals. We need to consider the following three cases:

(i) $\tau_j - m_j > C_U/2$. Clearly, $\tau_k - m_k > C_U/2$ for $k = 1, 2, \dots, j-1$. In other words, $\tau_i - m_i > C_U/2$ for $1 \leq i \leq j$. Hence

$$\begin{aligned} \sum_{k=1}^K (\tau_k - m_k) &> (Q+1)(\tau_j - m_j) > (Q+1) \frac{1}{2} C_U \\ &\geq (Q+1) \frac{1}{2} \frac{2}{Q+1} \sum_{k=1}^K (\tau_k - m_k) = \sum_{k=1}^K (\tau_k - m_k) \end{aligned}$$

We have a contradiction.

(ii) $\tau_j - m_j = C_U/2$. By the fact that R_j cannot fit in any of the Q intervals, more than $C_U/2$ units of time in each interval must have been assigned to tasks with total execution times exceeding $C_U/2$. Again,

$$\sum_{k=1}^K (\tau_k - m_k) > Q \frac{1}{2} C_U + \tau_j - m_j = (Q+1) \frac{1}{2} C_U \geq \sum_{k=1}^K (\tau_k - m_k)$$

and we have a contradiction.

(iii) $\tau_j - m_j < C_U/2$. Let $\tau_j - m_j = \lambda C_U$ for some fraction λ smaller than $1/2$. Since R_j cannot be assigned to any of the Q interval of length C_U , the total execution time of all tasks already assigned in every interval must exceed $(1-\lambda)C_U$. Moreover,

$$\sum_{k=1}^K (\tau_k - m_k) > Q(1-\lambda)C_U + \lambda C_U = \left[2(1-\lambda) - \frac{2-4\lambda}{Q+1} \right] \sum_{k=1}^K (\tau_k - m_k)$$

Since $2-4\lambda > 0$ and $Q \geq 2$, we have

$$\begin{aligned}
\sum_{k=1}^K (\tau_k - m_k) &\geq \left[2(1-\lambda) - \frac{2-4\lambda}{3} \right] \sum_{k=1}^K (\tau_k - m_k) \\
&= \left(1 - \frac{2\lambda-1}{3} \right) \sum_{k=1}^K (\tau_k - m_k) > \sum_{k=1}^K (\tau_k - m_k)
\end{aligned}$$

Again, we have a contradiction.

To show this upper bound is tight, we consider a remaining set \mathbf{R} consisting of $Q+1$ tasks. The execution time of every task is Δ . These $Q+1$ tasks can be scheduled in Q intervals with length 2Δ or larger. Since

$$C_U = \frac{2}{Q+1} \sum_{k=1}^{Q+1} \Delta = 2\Delta$$

the upper bound is tight. ■

Theorem V-2: A set \mathbf{J} of K type C jobs of repetition period p and cumulation rate $2Q-1$ is schedulable (i.e. a feasible schedule is guaranteed to be found) using the length monotone algorithm if

$$\frac{Q-1}{Q+1} u + \frac{2}{Q+1} U \leq 1$$

where u and U are total utilization factors of \mathbf{F} and \mathbf{J} , respectively, as defined in Eq. (9).

Proof: The proof of this theorem follows directly from Lemma V-1. The length of the Q intervals during which processor times can be assigned to tasks in \mathbf{R} is equal to $p - \sum_{k=1}^K m_k$. The set of tasks in \mathbf{R} can be scheduled precisely on non-preemptive basis if

$$p - \sum_{k=1}^K m_k \geq C_U = \frac{2}{Q+1} \sum_{k=1}^K (\tau_k - m_k)$$

Dividing both sides of the inequality by p , we have

$$1 \geq u + \frac{2}{Q+1} (U - u) = \frac{Q-1}{Q+1} u + \frac{2}{Q+1} U$$
■

The region of the $u-U$ plane in which the set J is schedulable is illustrated by Figure 6. The shaded area is the schedulable region for $Q = 2$ (i.e. cumulation rate 3).

V.2. Scheduling Simply Periodic Jobs

We consider a set J of simply periodic jobs consisting of subsets J_k for $k = 1, 2, \dots, K$; all jobs in J_k have the same repetition period p_k , ready times and deadlines, and $p_1 < p_2 < \dots < p_K = p$. Suppose that one task in every job must be completed at least once every $q p$ units of time. In other words, for every job with repetition period p_k , at least one task in every $(2q p / p_k) - 1$ consecutive periods must be completed. We say that these simply periodic jobs have the same cumulation rate of $q p$ units of time.

Figure 7 illustrates the problem of scheduling simply periodic jobs. In this example, there are five jobs to be scheduled. They are $J_1 = (1, 0.5)$, $J_2 = (1, 0.47)$, $J_3 = (1, 0.45)$, $J_4 = (2, 0.6)$ and $J_5 = (4, 0.7)$ and have minimal execution time 0.1. In term of the notation just introduced, $J = \{J_1, J_2, J_3\}$ where $J_1 = \{J_1, J_2, J_3\}$, $J_2 = \{J_4\}$ and $J_3 = \{J_5\}$. Suppose that $q = 1$, i.e. one task in every job must be completed every 4 units of time. We define the first set F as in Section III. The remaining set R consists of three subsets, $R_1 = \{R_1, R_2, R_3\}$, $R_2 = \{R_4\}$ and $R_3 = \{R_5\}$. Whenever a task in F is ready, it is scheduled according to the rate-monotone

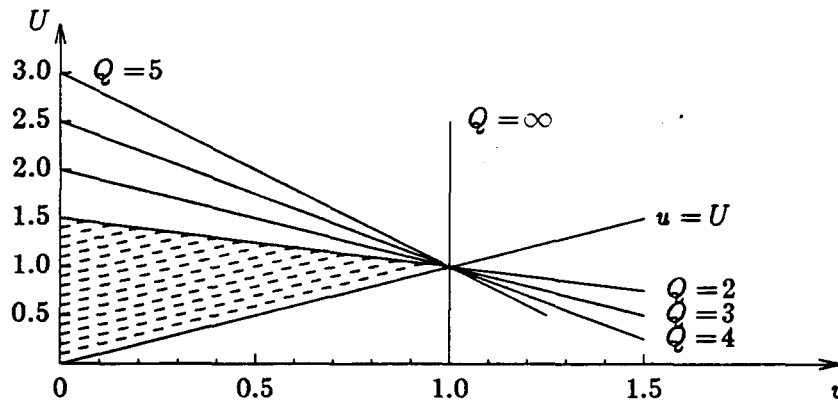


Figure 6. Schedulable region.

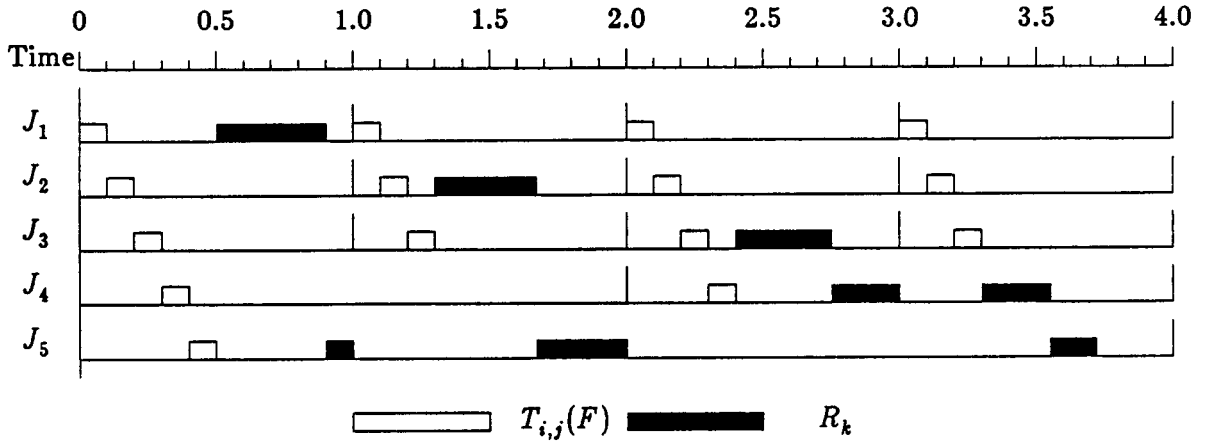


Figure 7. An example on scheduling of Type C simply periodic jobs.

algorithm. Tasks in \mathbf{R} are scheduled in the time intervals during which the processor is not assigned to tasks in \mathbf{F} . The lengths of these time intervals during the first 4 periods of J_1 are 0.5, 0.7, 0.6 and 0.7, respectively. The tasks R_1 , R_2 and R_3 in \mathbf{R}_1 have execution times 0.4, 0.37 and 0.35, respectively. Each of these tasks must be scheduled precisely in one of the 4 intervals so that it can execute from the start to finish in the interval. A reasonable heuristic is to assign the three tasks in \mathbf{R}_1 the highest priority among all tasks in \mathbf{R} by scheduling them first in these intervals using the FFD algorithm. As shown in Figure 7, R_1 , R_2 and R_3 are scheduled at 0.5, 1.3 and 2.4, respectively. R_4 with execution time 0.5 cannot fit in the first period of J_4 . It is scheduled at the intervals (2.75, 3) and (3.3, 3.25) in the second period of J_4 . Finally, R_5 is scheduled at the intervals (0.9, 1.0), (1.67, 2.0) and (3.55, 3.72) in a period of J_5 .

The above example suggests a natural way to extend the length monotone algorithm so that it can be used to schedule simply periodic jobs with the same cumulation rate. Again, the jobs in \mathbf{J} are divided into the first set \mathbf{F} and the remaining set \mathbf{R} . Tasks in \mathbf{F} are assigned higher priorities than the tasks in \mathbf{R} and are scheduled according to the rate-monotone algorithm. For each time interval of duration $q p$ and for each job J_i in \mathbf{J}_j with repetition period p_j , there is a task R_i with execution time $\tau_i - m_i$ in \mathbf{R}_j . This task must be scheduled precisely in one of the $q p / p_j$ periods of jobs in \mathbf{J}_j . The *extended length monotone algorithm* schedules the tasks in \mathbf{R} as follows: In the beginning of every period of J_1 , the processor is assigned to the ready tasks in \mathbf{F} .

The tasks in R_1 are then scheduled, making use of the spared processor time in $q p/p_1$ periods. Specifically, the FFD algorithm is used to assign the tasks in R_1 to the processor in the time intervals during which the processor is not busy executing tasks in F , and every task in R_1 is scheduled to fit within one period of J_1 . After the tasks in R_1 are scheduled, the spared processor time left in the $q p/p_2$ periods of J_2 is assigned to the tasks in R_2 , again, using the FFD algorithm. In general for all $k \leq K$, the tasks in R_k are scheduled in the $q p/p_k$ periods using the FFD algorithm after the tasks in R_1, R_2, \dots, R_{k-1} are scheduled.

Unfortunately, the amounts of spare processor time in different periods of each job are different. This fact makes it difficult to derive analytically the condition(s) under which a set of simply periodic jobs is guaranteed to be schedulable using the extended length monotone algorithm. A simulation study is being carried out to determine the performance of this algorithm.

VI. SUMMARY

This paper discusses the problem of scheduling periodic jobs on systems that allow tasks to terminate prematurely and produce imprecise results. When a task terminates normally, the error in the result produced by it is zero. When it terminates prematurely, the result produced by it is acceptable as long as the duration of its execution is equal to or longer than its minimal execution time. Hence, to guarantee that all deadlines are met requires only that the amount of processor time assigned to every task prior to its deadline is at least equal to its minimal execution time.

Depending on the undesirable effects caused by errors, jobs are classified as type N and type C. For type N jobs, the criterion used to measure the performance of scheduling algorithms is the average error over all jobs. Three heuristic algorithms that guarantee to produce feasible schedules of type N jobs are described. Their performance depends on how errors vary as functions of the assigned times of the tasks. The least utilization algorithm produces suboptimal schedules with small average error when errors in all jobs are linearly dependent on their assigned times. Bounds on the average error for schedules generated by this algorithm are

derived. Many iteration procedures converge faster during earlier iterations; the error is often a convex function of the assigned time. For this case, the least attained time algorithm should achieve less average error. The performance of the least attained time algorithm and the best slack time algorithm remains to be evaluated.

For type C jobs, the undesirable effects of errors produced in different periods are cumulative. We are concerned with the case when at least one task among every Q consecutive tasks in every job is required to complete normally. The length monotone algorithm can be used to schedule jobs with the same repetition period and cumulation rate. A necessary condition for a set of jobs to be schedulable using the length monotone algorithm is found. The way in which this algorithm can be used to schedule simply periodic type C jobs is discussed.

REFERENCES

- [1] Lin, K. J., S. Natarajan, J. W.-S. Liu, and T. Krauskopf, "Concord: a system of imprecise computations," *Proceedings of the 1987 IEEE Compsac*, Japan, October, 1987.
- [2] Lin, K. J., S. Natarajan, and J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," to appear in the *Proceedings of the IEEE Real-Time Systems Symposium*, 1987.
- [3] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," to appear in the *Proceedings of the IEEE Real-Time Systems Symposium*, 1987.
- [4] Liestman, A. L. and R. H. Campbell, "A fault-tolerant scheduling problem," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 11, pp. 1089-1095, November, 1986.
- [5] Basu, A. K., "On development of iterative programs from function specifications," *IEEE Transactions on Software Engineering*, Vol. SE-6, pp. 170-182, March, 1980.
- [6] Turski, W. M., "On programming by iterations," *IEEE Transactions on Software Engineering*, Vol. SE-10, pp. 175-178, March, 1984.
- [7] Chandy, K. M., J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144-156, May, 1983.
- [8] Bellman, R. and S. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, N. J., 1962.
- [9] Shreider, Y., ed., *Method of Statistical Testing: Monte Carlo Method*, Elsevier Publishing Company, New York, 1964.
- [10] Rabin, M. O., "Randomized Byzantine generals," *Proceedings 24th Symposium Foundations of Computer Science*, Tucson, Arizona, pp. 403-409, November, 1983.
- [11] Coffman, E. G. Jr. and R. Graham, *Scheduling Theory*, John Wiley and Sons, New York, 1976.
- [12] Lenstra, J. K. and A. H. G. Rinnooy Kan, "Scheduling theory since 1981: an annotated bibliography," Report No. BW 188/83, Mathematisch Centrum, Amsterdam, the Netherlands, 1983.
- [13] Lenstra, J. K. and A. H. G. Rinnooy Kan, "New directions in scheduling theory," *Operations Research Letters*, Vol. 2, pp. 255-259, 1984.
- [14] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, Vol. 20, No. 1, pp. 46-61, January, 1973.

- [15] Dhall, S. K. and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, Vol. 26, No. 1, pp. 127-140, 1978.
- [16] Leung, J. Y.-T. and M. L. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," *Information Processing Letters*, Vol. 11, No. 3, pp. 115-118, November, 1980.
- [17] Lawler, E. L. and C. U. Martel, "Scheduling periodically occurring tasks on multiple processors," *Information Processing Letters*, Vol. 12, No. 1, pp. 9-12, February, 1981.
- [18] Leung, J. Y.-T. and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, Vol. 2, pp. 237-250, 1982.
- [19] Liu, C. L., J. W. S. Liu and A. L. Liestman, "Scheduling with slack time," *Acta Informatica*, Vol. 17, pp. 31-41, 1982.
- [20] Bertossi, A. A. and M. A. Bonuccelli, "Preemptive scheduling of periodic jobs in uniform multiprocessor systems," *Information Processing Letters*, Vol. 16, pp. 3-6, January, 1983.
- [21] Stankovic, J., K. Ramamritham, and S. Chang, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, pp. 1130-1143, December, 1985.
- [22] Zhao, W. and K. Ramamritham, "Distributed scheduling using bidding and focused addressing," *Proceedings of IEEE Real-time Symposium*, December, 1985.
- [23] Leinbaugh, D. W. and M. Yamini, "Guaranteed response time in a distributed hard real-time environment," *Proceedings of Real-Time Systems Symposium*, pp. 157-169, December, 1982.
- [24] Wensley, J. H., L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, Vol. 66, No. 10, pp. 1240-1255, October, 1978.
- [25] Garey, M. R. and D. S. Johnson, "*Computers and intractability a guide to the theory of NP-Completeness*," W. H. Freeman and Company, New York, 1979.
- [26] Coffman, E. G. Jr., M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM J. Comput.*, Vol 7, No. 1, pp. 1-17, February, 1978.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1307	2.	3. Recipient's Accession No.
4. Title and Subtitle Scheduling Periodic Jobs Using Imprecise Results		5. Report Date November, 1987	
		6.	
7. Author(s) Jen-Yao Chung, Jane W. S. Liu, Kwei-Jay Lin		8. Performing Organization Repr. No.	
9. Performing Organization Name and Address Department of Computer Science University of Illinois 1304 West Springfield Urbana, Illinois 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No. NVY N00014 87 K 0827	
12. Sponsoring Organization Name and Address Office of Naval Research 800 North Quincey Road Arlington, Virginia 22217		13. Type of Report & Period Covered	
		14.	
15. Supplementary Notes			
16. Abstracts <p>One approach to avoid timing faults in hard, real-time systems is to make available intermediate, imprecise results produced by real-time processes. When a result of the desired quality cannot be produced in time, an imprecise result of acceptable quality produced before the deadline can be used. This paper discusses the problem of scheduling periodic jobs to meet deadlines on a system that provides the necessary programming language primitives and run-time support for processes to return imprecise results. Since the scheduler may choose to terminate a task before it is completed, causing it to produce an acceptable but imprecise result, the amount of processor time assigned to any task in a valid schedule can be less than the amount of time required to complete the task. A meaningful formulation of the scheduling problem must take into account the overall quality of the results. Depending on the different types of undesirable effects caused by errors, jobs are classified as type N or type C. For type N jobs, the effects of errors in results produced in different periods are not cumulative. A reasonable performance measure is the average error over all jobs. Three heuristic algorithms that lead to feasible schedules with small average errors are described. For type C jobs, the undesirable effects of errors produced in different periods are cumulative. Schedulability criteria of type C jobs are discussed.</p>			
17. Key Words and Document Analysis. 17a. Descriptors <p>Real-time systems scheduling to meet deadlines programming environment</p>			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
		20. Security Class (This Page) UNCLASSIFIED	22. Price